htmlChecker.py

A *markup language* is a system for inserting annotations into a document. The most important feature of a markup language is that the *tags* it uses to indicate annotations should be easy to distinguish from the document *content*.

One of the most well-known markup languages is the one commonly used to create Web pages, called **HTML**, or "Hypertext Markup Language". In HTML, tags appear in "angle brackets" such as in "<html>". When you load a Web page in your browser, you don't see the tags themselves: the browser interprets the tags as instructions on how to format the text for display.

Most tags in HTML are used in pairs to indicate where an effect starts and ends. For example:

- represents the start of a paragraph, and indicates where that paragraph ends.
- and are used to place the enclosed text in **bold** font, and and indicate that the enclosed text is *italic*.

Note that "end" tags look just like the "start" tags, except for the addition of a backslash after the < symbol.

Sets of tags are often nested inside other sets of tags. For example, an *ordered list* is a list of numbered bullets. You specify the start of an ordered list with the tag <ol>, and the end with . Within the ordered list, you identify items to be numbered with the tags <li> (for "list item") and . For example, the following specification:

```
<ol>
   <li>First item</li>
   <li>Second item</li>
   <li>Third item</li>
</ol>
```

would result in the following:

1. First item
2. Second item
3. Third item

Notice how you start the ordered list with the <ol> tag, specify three line items with matching <li> and tags, and the close the ordered list with the tag.

You may have noticed that the pattern of using matching tags strongly resembles the pattern of matching parentheses that we discussed in class: when you use parentheses, brackets, and braces, they have to match in reverse order, such as "{[()]}". A pattern such as "[(])" would be incorrect since the right bracket does not match the left parenthesis. Similarly, an HTML pattern such as would be incorrect since the closing tags are in the wrong order.

Your assignment is to write an "HTML Checker" program that takes as input an HTML file, and produces a report indicating whether or not the tags are correctly matched.

- Just as the parenthesis checker uses a stack to store symbols waiting for a match to be found, your program should also use a stack. **You should simply include the list implementation of the Stack abstract data type discussed in class in your code.**
- The first task your program must do is read in an HTML file and extract the tags. A simple strategy for doing this would be to write a function "getTag" that:
    - reads one character at a time from the data file, throwing everything away until it gets to a "<". (Discard the "<" as well.)
    - reads one character at a time, appending it to a string, until it gets to a ">" or whitespace. (Discard the ">" as well.)
    - returns the string as a tag.
- Call getTag repeatedly, saving the results in a list. Make sure you account for end-of-file conditions in getTag. If you've done everything correctly, you now have a list of tags, both start and end.
- Now go back and iterate through your list of tags, looking for matches:
    - if a tag does not start with a "/", it's a start tag. Push it on a stack.
    - if a tag starts with a "/", and the rest of it matches the tag on the stack, it's a correctly matched end tag. Pop the start tag off the stack.
    - if a tag starts with a "/" but does not match the tag on the top of the stack, there is a mismatch. Print an error message and terminate.

In addition, have your program build a list called "VALIDTAGS". As you iterate through your list of tags, check to see if the tag appears in VALIDTAGS. If it doesn't, add it to VALIDTAGS and print the message, "New tag XXX found and added to list of valid tags"

Input:

Use the input file given.

Output:

The output of your program should include the following:

- A printout of your list of tags (the result of your repeated calls to getTag).
- One line for each tag as you process it, explaining the action and showing the current contents of the stack. Some examples are:

```
Tag b pushed:  stack is now [html, body, b]
Tag /b matches top of stack:  stack is now [html, body]
Tag ul pushed:  stack is now [html, body, ul]
```

- A message every time you add a tag to VALIDTAGS.

Your program should end with either:

```
Error:  tag is XXX but top of stack is YYY
```

or

```
Processing complete.  No mismatches found.
```

or

```
Processing complete.  Unmatched tags remain on stack:  [XXX, YYY, ZZZ]
```

followed by labeled and sorted printouts of VALIDTAGS and EXCEPTIONS (see below).

Plot Twist:

There are some tags that do not need matching start and end tags! One example is
. This tag is used to indicate a line break at the current location. Another is <meta>, which is used to provide special information ("metadata") about a Web page, and one more (left for you to identify in your data file).

This means that if you followed the instructions above correctly, your HTML checker will notice that there are three tags that don't have a match. Teach your program that this is okay for these three cases by maintaining a list called EXCEPTIONS which you hard-code into your main program. They will appear in your list of tags just as any other tags. However, when you begin your iteration through the list and you encounter one of these, you do not need to push it on the stack since you won't be waiting for a close tag. Instead, just print an output line such as:

```
Tag br does not need to match:  stack is still [html, body, b]
```

and continue.